2014 IEEE Fourth International Conference on Big Data and Cloud Computing

On the Graph Decomposition

[1]Yangjun Chen and [2]Yibin Chen
Dept. Applied Computer Science, University of Winnipeg, Canada
[1]y.chen@uwinnipeg.ca, [2]chenyibin@gmail.com

***Abstract*—In this paper, we propose an efficient algorithm to decompose a directed acyclic graph (DAG) G into a minimized set of node-disjoint chains, which cover all the nodes of G. For any two nodes u and v on a chain, if u is above v then there is a path from u to v in G. The best algorithm for this problem up to now needs $O(n^3)$ time, where n is the number of the nodes of G. Our algorithm, however, needs only $O(max\{\sqrt{\kappa}\cdot\kappa n, \sqrt{n}\cdot m\})$ time, where n and m are the numbers of the nodes and arcs of G, and κ is G's width, defined to be the size of a largest node subset U of G such that for every pair of nodes $x, y \in U$, there does not exist a path from x to y or from y to x. κ is in general much smaller than n. In addition, by the existing algorithm, $\Theta(n^2)$ extra space (besides the space for G itself) is required to maintain the transitive closure of G to do the task while ours needs only $O(\kappa n)$ extra space. Considering the nowadays applications with massive graphs including millions and even billions of nodes, like the *facebook* and *twitter*, space reduction is also very important.**

Key words:* *reachability queries, directed graphs, transitive closure, graph decomposition

I. INTRODUCTION

Let G be a directed acyclic graph (a *DAG* for short). A *chain cover* of G is a set C of *node-disjoint chains* such that it covers all the nodes of G, and for any two nodes u and v on a chain $p \in C$, if u is above v then there is a path from u to v in G. In this paper, we discuss an efficient algorithm to find a *minimized* C for G. As an example, consider the DAG shown in Fig. 1(a). We can decompose it into a set of two chains, as shown in Fig. 1(b), which covers all the nodes of G. Fig. 1(c) shows another possible minimized decomposition.

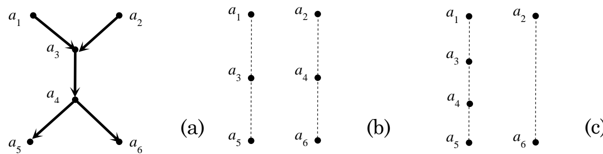


Fig. 1. Illustration for DAG decomposition

With the advent of the web technology, the efficient decomposition of a DAG G into a minimum set of chains becomes very important; especially, for the applications involving massive graphs such as social networks, for which we may quite often ask whether a node v is reachable from another node u through a path in G. A naive method to answer such a query is to precompute the reachability between every pair of nodes in $G(V, E)$ - in other words, to compute the transitive closure of G, which is also a directed graph $G^*(V, E^*)$ with $(v, u) \in E^*$ iff there is a path from v to u in G. (See Fig. 2(a) for illustration, in which we show the transitive closure of the graph shown in Fig. 1(a).)

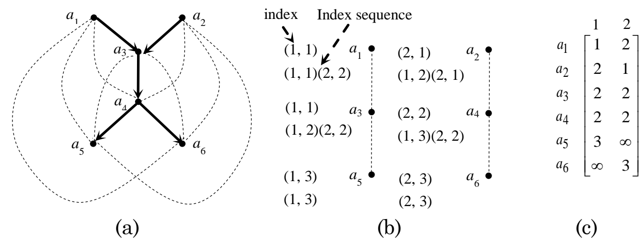


Fig. 2. Illustration for transitive closure and index

As it is well known, the transitive closure of G can be stored as a boolean matrix $\boldsymbol{M}$ such that $\boldsymbol{M}[i, j] = 1$ if there is path from i to j; otherwise, $\boldsymbol{M}[i, j] = 0$ [18]. Then, a reachability query can be answered in a constant time. However, this requires $O(n^2)$ space for storage, which makes it impractical for very large graphs, where $n = |V|$. Another method is to compute the shortest path from u to v over such a large graph on demand. Therefore, it needs only $O(m)$ space, but with high query processing cost - $O(m)$ time in the worst case, where $m = |E|$. However, if we are able to decompose a DAG into a minimum set of chains, we can effectively compress a transitive closure without increasing much query time, as described below.

Let G be a directed graph. If it is cyclic (i.e., it contains cycles), we can first find all the *strongly connected components* (*SCC*) in linear time [17] and then collapse each of them into a representative node. Clearly, all of the nodes in an SCC are equivalent to its representative as far as reachability is concerned since each pair of nodes in an SCC are reachable from each other. In this way, we transform G to a DAG. Next, we decompose the DAG into a minimum set C of node-disjoint chains. (Recall that if a node u appears above another node v on a chain, there is a path from u to v.) Denote $|C| = \kappa$. We will then

(1) number each chain and number each node on a chain; and
(2) use a pair (i, j) as an index for the jth node on the ith chain.

Besides, each node u on a chain will be associated with an index sequence of the form: $(r, j_r) \dots (i, j_i) \dots (k, j_k)$ $(1 \le r \le i \le k \le \kappa)$ such that any node v with index (x, y) is a descendant of u iff there exists (x, j_x) in the sequence with $y \ge j_x$. (See Fig. 2(b) for illustration.) Such index sequences can be created as follows.

First of all, we notice that we can associate each leaf node with an index sequence, which contains only one index, i.e., the index assigned to it. Clearly, such an index sequence is trivially sorted and its length is $1 \le \kappa$. Let v be a non-leaf node with children $v_1, \dots, v_l$ each associated with an index sequence

978-1-4799-6719-3/14 $31.00 © 2014 IEEE
DOI 10.1109/BDCloud.2014.118

IEEE computer society

L_i ($1 \leq i \leq l$). Assume that $|L_i| \leq \kappa$ ($1 \leq i \leq l$) and the indexes in each L_i are sorted according to the first element in each index. We will create an index sequence L for v, which initially contains only the index assigned to it. Then, we will merge all L_i's into L one by one. To merge an L_i into L, we will scan both L and L_i from left to right. Let (a_1, b_1) (from L) and (a_2, b_2) (from L_i) be the index pairs currently encountered. We will perform the following checkings:

- If $a_2 > a_1$, we go to the index next to (a_1, b_1) (in L) and compare it with (a_2, b_2) in a next step.
- If $a_1 > a_2$, insert (a_2, b_2) just before (a_1, b_1) (in L). Go to the index next to (a_2, b_2) (in L_i) and compare it with (a_1, b_1) in a next step.
- If $a_1 = a_2$, we will compare b_1 and b_2. If $b_1 < b_2$, nothing will be done. If $b_2 < b_1$, replace b_1 (in (a_1, b_1)) with b_2. In both cases, we will go to the indexes next to (a_1, b_1) (in L) and (a_2, b_2) (in L_i), respectively.
- We will repeat the above three steps until either L or L_i is exhausted. If when L is exhausted L_i still has some remaining elements, append them at the end of L.

Obviously, after all L_i's have been merged into L, the length of L is still bounded by the number κ. Denote by d_v the outdegree of v. The time spent on this process is then bounded by $O(\sum_v d_v \cdot \kappa) = O(\kappa \cdot m)$, but the space overhead is only $O(\kappa \cdot n)$. The query time remains O(1) if we store the index sequences as a matrix $\boldsymbol{M}_G$, as shown in Fig. 2(c), in which each entry $\boldsymbol{M}_G(v, j)$ is the jth element in the index sequence associated with node v. So, a node u with index (i, j) is a descendant of node v iff $\boldsymbol{M}_G(v, i) \leq j$. In practise, κ is in general much smaller than n. In this sense, G^* is effectively compressed based on a minimized decomposition of G.

The problem to decompose a DAG is also heavily related to another theoretical problem: the decomposition of *partially ordered sets* (or *posets* for short) $\boldsymbol{S} = (S, \succeq)$ into a minimum set of chains, where S is a set of elements and $\succeq$ is a *reflexive*, *transitive*, and *antisymmetric* relation over the elements [7].

In [12], Jagadish discussed an algorithm for finding a minimum set of node-disjoint paths that cover a directed acyclic graph G by transforming the problem to a *min* network flow [8, 15]. Its time complexity is bounded by $O(n \cdot m)$. But a chain is in general not a path. For any pair of nodes u and v on a chain, we only require that if u appears above v, there is a path from u to v. So, the number of paths found by the method discussed in [12] is generally much larger than the minimal number of node-disjoint chains. However, if we apply the Jagadish's method to G^*, we can get a minimized set of chains for G. But again, $O(n^3)$ time and $\Theta(n^2)$ space are required to construct G^*.

The method discussed in [3] is also to decompose a DAG into node-disjoint chains. It runs in $O(n^{2.5})$ time. However, the decomposition found is not minimum. Our earlier algorithm [4] works for the same purpose. Its time complexity is bounded by $O(k^{1.5}n)$, where k is the number of the chains, into which a DAG is decomposed. But in some cases it fails to find a minimum set of chains since when generating chains, only part of reachability information is considered. This problem is removed by [5] and [6] both with the same time complexity $O(\kappa \cdot n^2)$. However, in the method discussed in [5] each node is associated with a large data structure and requires $O(\kappa \cdot n^2)$ space in the worst case. By [6], the generated chains may contain some newly created nodes, but how to remove such nodes are not discussed at all.

Different from the above strategies, the algorithm discussed in [9] is to find a maximum k-chain in a planar point set $M \subseteq N \times N$, where $N = \{0, 1, ..., n - 1\}$ and is defined by establishing $(i', j') \succ (i, j)$ iff $i' > i$ and $j' > j$. So M is a special kind of posets. A k-chain is a subset of M that can be covered by k chains. The time complexity of this algorithm is bounded by $O((n^2/k)/\log n)$. The algorithms discussed in [13] and [16] are to find a maximum 2-chain and 1-chain in M, respectively. [13] needs $\Theta(n \cdot \log n)$ time while [16] needs only $O(p \cdot n)$ time, where p is the length of the longest chain.

In this paper, we propose an efficient algorithm to find a minimum set of chains for G. It runs in $O(max\{\sqrt{\kappa} \cdot \kappa \cdot n, \sqrt{n} \cdot m\})$ time and in $O(\kappa \cdot n)$ space while the best algorithm for this problem needs $O(n^3)$ time and $\Theta(n^2)$ space.

The remainder of the paper is organized as follows. In Section 2, we discuss an algorithm to stratify a DAG into different levels. Section 3 is devoted to the description of our algorithm to decompose a DAG into chains, as well as the analysis of its computational complexities. We conclude our paper in Section 4.

II. GRAPH STRATIFICATION AND BIPARTITE GRAPHS

Our method is based on a DAG stratification strategy and an algorithm for finding a maximal matching in a bipartite graph. We first discuss the DAG stratification.

Definition 1 Let $G(V, E)$ be a DAG. We decompose V into subsets $V_0, V_1, ..., V_h$ such that $V = V_0 \cup V_1 \cup ... \cup V_h$ and each node in V_i has its children appearing only in $V_{i-1}, ..., V_0$ ($i = 1, ..., h$), where h is the height of G, i.e., the length of the longest path in G. □

For each node v in V_i, we say, its level is i, denoted $level(v) = i$. We also use $C_j(v)$ ($j < i$) to represent a set of links which start from v to all those v's children, which appear in V_j. Therefore, for each v in V_i, there exist $i_1, ..., i_k$ ($i_l < i$, $l = 1, ..., k$) such that the set of its children equals $C_{i_1}(v) \cup ... \cup C_{i_k}(v)$. Let $V_i = \{v_1, v_2, ..., v_l\}$. We use $\boldsymbol{C}_j^i$ ($j < i$) to represent $C_j(v_1) \cup ... \cup C_j(v_l)$.

Such a DAG decomposition can be done in $O(m)$ time by using the following algorithm, in which we use $G_1 \backslash G_2$ to stand for a graph obtained by deleting the arcs of G_2 from G_1; and $G_1 \cup G_2$ for a graph obtained by adding the arcs of G_1 and G_2 together. In addition, $d_{in}(v)$ and $d_{out}(v)$ represent v's indegree and v's outdegree, respectively.

In the above algorithm, we first determine V_0, which contains all those nodes having no outgoing arcs (see line 1). In the subsequent computation, we determine $V_1, ..., V_h$. In this process, G is reduced step by step (see line 8), so is $d_{out}(v)$ for any $v \in G$ (see line 9). In order to determine V_i ($i > 0$), we will first find all those nodes that have at least one child in V_{i-1}, which are stored in a temporary variable W. For each node v in W (see line 3), we will then check whether it also has some other children not appearing in V_{i-1}, which is done by checking whether $d_{out}(v) > k$ in line 7, where k is the number of v's

children in V_{i-1}. If it is the case, it will be removed from W since it cannot belong to V_i. Concerning the correctness of the algorithm, we have the following proposition.

ALGORITHM 1. *GraphStra*(G)

Begin

1. V_0 := all the nodes with no outgoing arcs; i := 0;
2. W := all the nodes that have at least one child in V_0;
3. **while** $W \neq \emptyset$ **do**
4.   **for** each node v in W **do**
5.   **let** $v_1, ..., v_k$ be v's children appearing in V_i;
6. $C_i(v) := \{v_1, ..., v_k\}$; (*Here, for simplicity, we use v_j to represent a link from v to v_j.*)
7.   **if** $d_{out}(v) > k$ **then** remove v from W;
8. $G := G \backslash \{v \to v_1, ..., v \to v_k\}$;
9. $d_{out}(v) := d_{out}(v) - k$;
10. $V_{i+1} := W$; $i := i + 1$;
11. W := all the nodes that have at least one child in V_i;

end

In the above algorithm, we first determine V_0, which contains all those nodes having no outgoing arcs (see line 1). In the subsequent computation, we determine $V_1, ..., V_h$. In this process, G is reduced step by step (see line 8), so is $d_{out}(v)$ for any $v \in G$ (see line 9). In order to determine V_i ($i > 0$), we will first find all those nodes that have at least one child in V_{i-1}, which are stored in a temporary variable W. For each node v in W (see line 3), we will then check whether it also has some other children not appearing in V_{i-1}, which is done by checking whether $d_{out}(v) > k$ in line 7, where k is the number of v's children in V_{i-1}. If it is the case, it will be removed from W since it cannot belong to V_i.

Since each arc is accessed only once in the process, the time complexity of the algorithm in bounded by O(m).

As an example, consider the graph shown in Fig. 3(a). Applying the above algorithm to this graph, we will generate a stratification of the nodes as shown in Fig. 3(b).

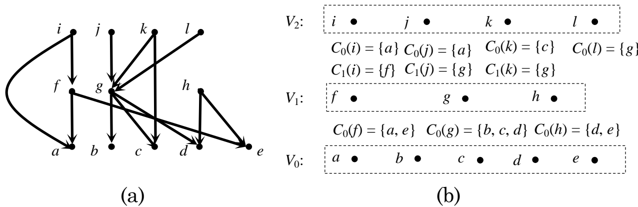


Fig. 3. Illustration for DAG stratification

In Fig. 3(b), the nodes of the DAG shown in Fig. 3(a) are divided into three levels: $V_0 = \{a, b, c, d, e\}$, $V_1 = \{f, g, h\}$, and $V_2 = \{i, j, k, l\}$. Associated with each node at each level is a set of links pointing to its children at different levels. For example, node g in V_1 is associated with three links respectively to nodes b, c, and d in V_0, denoted as $C_0(g) = \{b, c, d\}$. (For simplicity, we use $C_0(g) = \{b, c, d\}$ to represent three links from g to b, c, and d, respectively.)

III. ALGORITHM DESCRIPTION

In this section, we describe our algorithm for the DAG decomposition. The main idea behind it is to construct a series of bipartite graphs for $G(V, E)$ based on the graph stratification and then find a maximum matching for each of such bipartite graphs using the Hopcroft-Karp algorithm [11]. All these matchings make up a set of node-disjoint chains, which, however, may not be minimal. In the following, we first discuss an example to illustrate this idea in Subsection *A*. Then, in Subsection *B*, we define the so-called *virtual nodes*, and show how they can be used to efficiently and effectively reduce the number of node-disjoint chains. Next, in Subsection *C*, we discuss how the virtual nodes can be resolved (removed) from created chains to get the final result.

A. Chain Generation

From the above example, we can see that by simply combining maximal matchings of bipartite graphs, the number of formed chains may be larger than the minimized number of chains. To solve this problem, we need to introduce some virtual nodes into the original graph, which are used to transfer the reachability information from lower levels to higher levels.

1) Basic idea: virtual nodes

We will work bottom-up. During the process, some virtual nodes may be added to V_i (i = 1, ..., h - 1) level by level. However, such virtual nodes will be eventually resolved to obtain the final result.

In the following, we first give a formal definition of virtual nodes. Then, we describe how a virtual node is established. We start our discussion with the following specification:

$V_0' = V_0$.

$V_i' = V_i \cup$ {virtual nodes added to V_i} for $1 \leq i \leq h - 1$.

$\boldsymbol{C}_i = \boldsymbol{C}_{i-1}^{i} \cup$ {all the new arcs from the nodes in V_i to the virtual nodes added to V_{i-1}'} for $1 \leq i \leq h - 1$.

$B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ - the bipartite graph containing V_i and V_{i-1}'.

M_i - a maximal matching of $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$.

Definition 2 (*virtual nodes*) Let $G(V, E)$ be a DAG, divided into $V_0, ..., V_h$ (i.e., $V = V_0 \cup ... \cup V_h$). Let M_i be a maximal matching of $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$ for i = 1, …, h. For each free node v in V_{i-1}' with respect to M_i, a virtual node v' created for v is a new node added to V_i ($1 \leq i \leq h - 1$), denoted as $v = s(v')$. □

The goal of virtual nodes is to establish the connection between the free nodes (with respect to a certain maximum matching of a bipartite graph) and the nodes that may be several levels apart. Therefore, for each virtual node v' (created for v in V_{i-1}' and added to V_i), a bunch of virtual arcs incident to it should be created. Especially, we distinguish among three kinds of virtual arcs, which are created in different ways:

inherited arcs - If there is $u \in V_j$ ($j > i$) such that $u \to v \in E$, add $u \to v'$, referred to as an inherited arc.

transitive arcs - If there exist $u \in V_j$ ($j > i$) and $w \in V_i$ such that $u \to w \in E$ and $w \to v \in \boldsymbol{C}_i$, add $u \to v'$ if it has not been created as an inherited arc, referred to as a transitive arc.

alternating arcs of the first kind - If there exists a node $w \in V_{i-1}'$ (covered by M_i) such that one of v's parents is connected to w through an α-segment (which is an *alternating path* with the edges in M_i and the edges not in M_i interleaved, starting and ending both at a by M_i covered edge) in $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$, and $u \in V_j$ ($j > i$) such that one of the two conditions holds:

- $u \to w \in E$, or

- there is a node $x \in V_i$ such that $u \to x \in E$ and $x \to w \in \boldsymbol{C}_i$,

add $u \to v'$ if it has not been created as an inherited or a transitive arc. It is referred to as an alternating arc of the first kind. We create such an arc to indicate a possibility to make v covered by transferring the edges on the corresponding alternating path from v to w, and then connect u and w.

In addition, a virtual arc from v' to $s(v')$ is generated to record the relationship between v' and $s(v')$.

Example 1 Continued with Fig. 3. Relative to M_1 of $B(V_1, V_0; E_1)$ shown in Fig. 4(a), c and e are two free nodes. Then, two virtual nodes c' and e' (for c and e, respectively) will be created and added to V_1. Then, we have $V_1' = \{f, g, h, c', e'\}$. In addition, seven virtual arcs: $i \to e'$, $j \to c'$, $j \to e'$, $k \to c'$, $k \to e'$, $l \to c'$, and $l \to e'$ will be generated, shown as eight dashed arcs in Fig. 4(b).

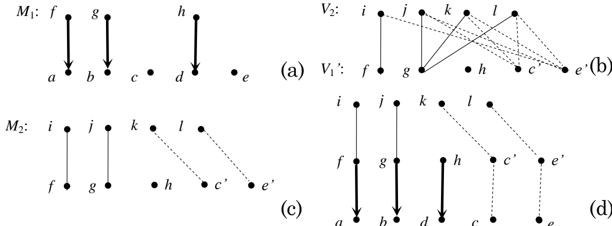


Fig. 4. Illustration for virtual nodes and chains with virtual nodes

Among these virtual arcs, $k \to c'$ is an inherited arc since in the original graph we have $k \to c$ (see Fig. 3(a)). But $j \to c'$, $l \to c'$, and $i \to e'$ are three transitive arcs since c is reachable respectively from j and l through g in V_1, and e is reachable from i through f in V_1. (see Fig. 3(a)).

Finally, $j \to e'$, $k \to e'$ and $l \to e'$ are three alternating arcs. We join j to e' since there is a node b that is connected to e's parent h through an α-segment: $b — g — d — h$ (in $B(V_1, V_0, E_1)$) and b is reachable from j in G through a node g (in V_1) (see Fig. 3(a).) For the same reason, we join k to e', and l to e'.

In Fig. 4(c), we show a possible maximum matching M_2 of $B(V_2, V_1'; \boldsymbol{C}_2)$. Combining M_2 and M_1, we get a set of five chains as shown in Fig. 4(d). Of the two virtual nodes c' and e', c' can be simply removed and connect k to c since $k \to c'$ is an inherited arc. In order to remove e', we have to transfer the edges on the alternating path: $b — g — d — h — e$ and then connect l and b, obtaining the final set of 5 chains.

We will call an arc along a chain a *chain arc*. From the above example, we can see that how a virtual node is resolved depends on how it is connected to its parent through a chain arc. Especially, an alternating arc in fact does not represent a reachability, but indicates a possibility to connect two nodes by transferring edges along some alternating path. Thus, we need to label virtual arcs to represent their properties, and at the same time indicate at what level a virtual node is added. Let v' be a virtual node. Depending on whether its source $s(v')$ is an actual node or a virtual node itself, we label the virtual arcs incident to v' in two different ways.

Assume that $s(v')$ is an actual node in V_{i-1}. Then, v' is a virtual node added to V_i and an virtual arc incident to v': $u \to v'$ with $u \in V_j$ ($j > i$) will be labeled as follows:

i) If $u \to v'$ is inherited or transitive, its label $label(u \to v')$ will be set to 0, indicating that $s(v')$ is reachable from u (through a path in G).

ii) If $u \to v'$ is an alternating arc, $label(u \to v')$ will be set to i, indicating that to resolve v' we need to transfer edges along an alternating path in $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$.

If $s(v')$ itself is a virtual node, we need to label $u \to v'$ a little bit differently:

iii) If $u \to v'$ is inherited (i.e., $u \to s(v')$ already exists), the label for it is set to be the same as $label(u \to s(v'))$.

iv) If $u \to v'$ is transitive, there must exist $w_1, ... w_k$ ($k \geq 1$) in V_i such that $w_1 \to s(v'), ..., w_k \to s(v') \in \boldsymbol{C}_i$ and $u \to w_1, ..., u \to w_k \in E$. We will label $u \to v'$ with $min\{l_1, ..., l_k\}$, where $l_j = label(w_j \to s(v'))$ ($j = 1, ..., k$).

v) If $u \to v'$ is an alternating arc, $label(u \to v')$ is set to i (in the same way as (ii)).

In addition, for convenience, all the original arcs in G are considered to be labeled with 0.

In the whole process, we will not only create a set of chains which may contain virtual nodes, but also a new graph by adding virtual nodes and virtual arcs to G, called a *companion graph* of G, denoted as G_c, which will be used for resolution of virtual nodes.

Example 2 Consider the graph shown in Fig. 5(a). This graph can be divided into five levels as shown in Fig. 5(b).

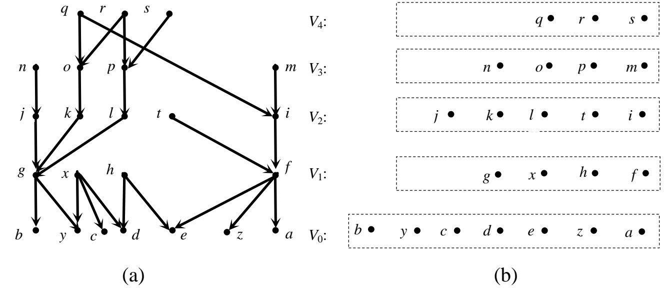


Fig. 5. A DAG and its stratification

In Fig. 6(a), we show the bipartite graph $B(V_1, V_0; \boldsymbol{C}_1)$ made up of the first two levels. A possible maximal matching M_1 of it is shown in Fig. 6(b). Relative to M_1, c, e and z are three free nodes in V_0. So three virtual nodes c', e' and z' will be created and added to V_1. At the same time, 15 arcs will be created, as shown in Fig. 6(c).

Among them, there are four transitive arcs: $t \to e'$, $t \to z'$, $i \to e'$, $i \to z'$; six alternating arcs: $j \to c'$, $j \to e'$, $k \to c'$, $k \to e'$, $l \to c'$, $l \to e'$.

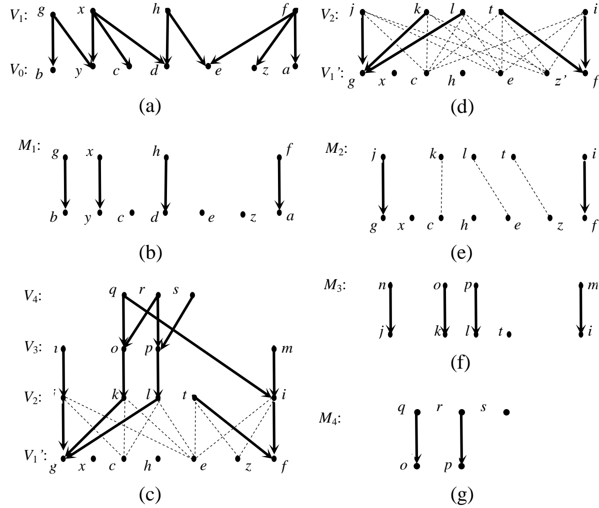
We have the transitive arc $t \to e'$ since e is reachable from t in G through a node f in V_1. The same claim applies to the other three transitive arcs.

The alternating arc: $j \to c'$ is created since there is an alternating path $c - x - y - g - b$ in $B(V_1, V_0; \boldsymbol{C}_1)$ and b is reachable from j in G. In a similar way, we can analyze all the other five alternating arcs of the first kind.

Thus, $V_1' = \{c', e', z', g, x, h, f\}$. $B(V_2, V_1'; \boldsymbol{C}_2)$ is shown in Fig. 6(d). Assume that the maximal matching M_2 found for it is as shown in Fig. 6(e), and M_3 for $B(V_3, V_2'; \boldsymbol{C}_3)$ and M_4 for

$B(V_4, V_3'; \boldsymbol{C}_4)$ are as shown in Fig. 6(f) and 6(g), respectively. By combining M_1, M_2, M_3 and M_4, we get $M_1 \cup M_2 \cup M_3 \cup M_4$. This plus all the free nodes in V_4 make up a set of eight chains as shown in Fig. 7(a), and one of them contains only a single node.

We can simply connect t and z' since $t \to z'$ is a transitive arc. We can also transfer the edges on P_1 and then connect k and b as shown in Fig. 7(b). After that, removing e' will leave l and e disconnected, resulting in a set of nine chains. It is not minimum. In Fig. 7(c), we show a possible decomposition of eight chains. In the next subsection, we discuss how the problem can be figured out.



$label(t \to e') = 0$ $label(t \to z') = 0$ $label(i \to e') = 0$ $label(i \to z') = 0$ $label(j \to c') = 1$
$label(j \to e') = 1$ $label(k \to c') = 1$ $label(k \to e') = 1$ $label(l \to c') = 1$ $label(l \to e') = 1$

Fig. 6. Illustration for generation of chains

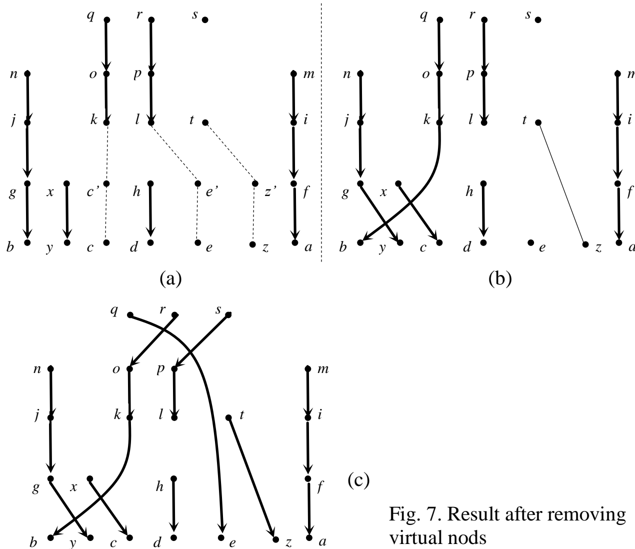


Fig. 7. Result after removing virtual nods

2) General algorithm for chain generation

To solve the above problem, we need to slightly modify the working process. For this, we need a new concept.

Definition 3 (*alternating graph*) Let $B(T, S; E)$ be a bipartite graph. Let M be a matching of $B(T, S; E)$. The alternating graph $\vec{B}$ with respect to M is a directed graph with the following sets of nodes and arcs:

$$\vec{V} = V(\vec{B}) = T \cup S, \text{ and}$$

$$\vec{E} = E(\vec{B}) = \{u \to v \mid u \in S, v \in T, \text{ and } (u, v) \in M\} \cup \{v \to u \mid u \in S, v \in T, \text{ and } (u, v) \in E \backslash M\}. \square$$

In Fig. 8(a), we show the alternating graph $\vec{B}_1$ with respect to M_1 for $B(V_1, V_0, \boldsymbol{C}_1)$ shown in Fig. 6(a). Assume that the maximum matching M_2 for $B(V_2, V_1, \boldsymbol{C}_2)$ is as shown in Fig. 8(b). Then, the corresponding alternating graph is a graph shown in Fig. 8(c).

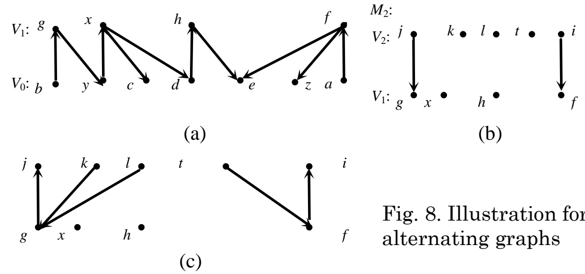


Fig. 8. Illustration for alternating graphs

Next, we will combine two consecutive alternating graphs $\vec{B}_i = \vec{B}(V_i', V_{i-1}'; \boldsymbol{C}_i)$ and $\vec{B}_{i+1} = \vec{B}(V_{i+1}, V_i'; \boldsymbol{C}_{i+1})$, denoted as $\vec{B}_i \oplus \vec{B}_{i+1}$, by connecting each node in V_{i+1} to all its reachable nodes in V_{i-1}'. In Fig. 9(a), we show $\vec{B}_1 \oplus \vec{B}_2$ for the graph shown in Fig. 5(a). We notice that in $\vec{B}_1 \oplus \vec{B}_2$, the nodes in V_1 are stored two times and the copy of a node v is considered to be a different node from v.

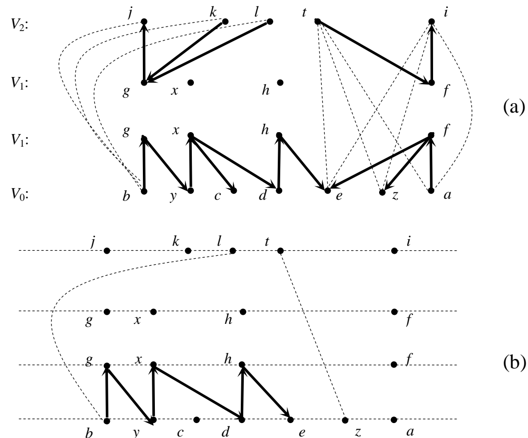


Fig. 9. Illustration for combined graphs and node-disjoint paths

What we want is to find a maximum set Γ of node-disjoint paths in $\vec{B}_i \oplus \vec{B}_{i+1}$, each starting from a free node u relative to M_{i+1} in V_{i+1}, and ending at a free node v relative to M_i in V_{i-1}', Let P be such a path which can always be divided into two parts: P' and P'' such that P' contains only the nodes in $\vec{B}_i$ while P'' contains only the nodes in $\vec{B}_{i+1}$. We will create a virtual node v' for v, connect it to the last node on P', and then

transfer the edges on P'. However, for each free node (in $V_{i\text{-}1}'$) not appearing on such a path, its virtual node will be added to V_{i+1}, for which only inherited and transitive arcs, as well as a new kind of virtual arcs, called supplementary arcs will be created.

alternating arcs of the second kind – Let v' be a virtual node created for v in $V_{i\text{-}1}'$ and added to V_{i+1}. If there exist a free node $w \in V_{i\text{-}1}'$ (relative to M_i) and a node $u \in V_j$ ($j > i$) such that one of v's parents is connected to w through a β-segment (which is an *alternating path*, starting and ending both at an edge not covered by M_i) in $B(V_i', V_{i\text{-}1}'; \boldsymbol{C}_i)$, satisfying one of the following two conditions:

- $u \rightarrow w \in E$, or
- there is an alternating path in $\vec{B}$ (V_i', $V_{i\text{-}1}'$; $\boldsymbol{C}_i$), which does not go through any node in Γ, but connects w to a node $x \in V_{i\text{-}1}'$ such that x is reachable from u,

add $u \rightarrow v'$ if it has not been created as an inherited or a transitive arc. $label(u \rightarrow v')$ is set to be i, same as an alternating arc incident to a virtual node added to V_i'. We create such an arc to resolve the conflict among free nodes in the case that they share a same alternating path P to a certain node. In this case, one free node, for example, node w can get covered by transferring the edges on P. But some other free node v which shares P with w may still be able to get covered along P if it is possible to make w covered along a different alternating path. To see this, let's check $\vec{B}_1 \oplus \vec{B}_2$ shown in Fig. 9(a) again, in which we can find a maximum set of two paths: $P_1 = l \rightarrow b \rightarrow g \rightarrow y \rightarrow x \rightarrow d \rightarrow h \rightarrow e$ and $P_2 = t \rightarrow z$ as shown in Fig. 9(b). Then, we add two virtual node e' and z' to V_1 and a virtual node c' to V_2. Especially, P_1 can be divided into $P_1' = l$ and $P_1'' = b \rightarrow g \rightarrow y \rightarrow x \rightarrow d \rightarrow h \rightarrow e$; and P_2 into $P_2' = t$ and $P_2'' = z$. So e' will be connected to l according to P_1, and z' will be connected to t according to P_2. Furthermore, c' will be connected to m and q for the following reason:

- there is a free node e in V_1 which is connected to c's parent x through a β-segment: $x - d - h - e$, and
- e is reachable from both m and q in G.

See Fig. 10(a) for illustration.

In a next step, we will consider V_1', V_2', V_3 and determine new virtual nodes to be added to V_2' and V_3, by which any node in V_1', which does not have parents needn't be considered any more. Assume that the maximum matching M_2 found for $B(V_2, V_1; \boldsymbol{C}_2)$ is as shown in Fig. 9(b). With virtual nodes e', z' and c' added, M_2 is extended as illustrated in Fig. 10(b). There is no free node in V_1' relative to M_2, and thus no new virtual nodes will be added to V_2'. Finally, we will consider V_2', V_3', V_4. Assume that the maximum matching M_3 found for $B(V_3, V_2'; \boldsymbol{C}_3)$ is as shown in Fig. 10(c). Then, c' is a free node in V_2' relative to M_3. We continually assume that the maximum matching M_4 found for $B(V_4, V_3'; \boldsymbol{C}_4)$ is as shown in Fig. 10(d). A maximum set of node-disjoint paths in $\vec{B}_3 \oplus \vec{B}_4$ contains only one path: $P = s \rightarrow p \rightarrow r \rightarrow o \rightarrow q \rightarrow i \rightarrow m \rightarrow c'$, which can be divided into $P' = s \rightarrow p \rightarrow r \rightarrow o \rightarrow q$ and $P'' = i \rightarrow m \rightarrow c'$. So the virtual node c'' created for c' will be connected to q, as demonstrated in Fig. 10(e). (We notice that t does not have parents and therefore no virtual node for it will be generated.) Transferring edges on P', we will change M_4 to a matching as shown in Fig. 11(a), and the final chains $M_1 \cup M_2 \cup M_3 \cup M_4$ is as shown in Fig. 11(b).

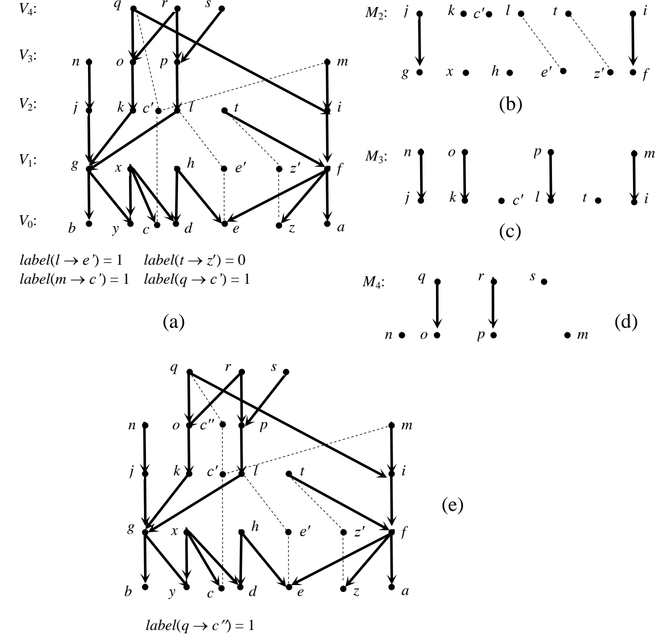


Fig. 10. Illustration for generation of virtual nodes

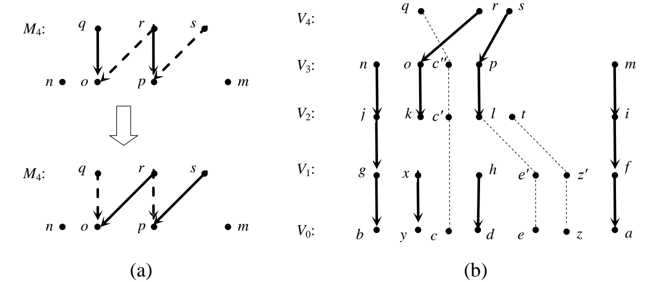


Fig. 11. Illustration for generating virtual nodes

According to the above discussion, we design a process, denoted as *VirtualGen*($V_{i\text{-}1}'$, V_i', V_{i+1}, M_i), conducting the following task:

1. It takes $V_{i\text{-}1}'$, V_i', V_{i+1}, M_i as the input.
2. Find M_{i+1} and form $\vec{B}_i \oplus \vec{B}_{i+1}$. Find a maximum set of node-disjoint paths in $\vec{B}_i \oplus \vec{B}_{i+1}$, each starting from a free node u relative to M_{i+1} in V_{i+1}, and ending at a free node v relative to M_i in $V_{i\text{-}1}'$. For each free node in $V_{i\text{-}1}'$ appearing on a path in this set, the created virtual node v' is added to V_i'. For each free node in $V_{i\text{-}1}'$ not appearing on a path in this set, the created virtual node v' is added to V_{i+1}. Create virtual arcs as described above.
3. M_{i+1} is used as the output of the process.

Based on this process, the general algorithm for the chain generation can be formally described as below.

ALGORITHM 2. *GenChain*(*stratification of G*)

input: a graph stratification.
output: a set of chains which may contain virtual nodes.
begin

1. $V_0' := V_0$; $V_1' := V_1$;
2. find M_1 for $B(V_1, V_0; \mathbf{C}_1)$;
3. **for** $i = 1$ to $h - 1$ **do**
4. { $M_{i+1} = VirtualGen(V_{i-1}', V_i', V_{i+1}, M_i)$; }
5. $M := M_1 \cup \ldots \cup M_h$; return M;

end

In the above algorithm, special attention should be paid to lines 1 - 2, by which the input for the first call of *VirtualGen*( ) is prepared. In the main **for**-loop, the input for a next call of *VirtualGen*( ) is produced in the current execution of *VirtualGen*( ).

We also notice that at any point in time only the virtual nodes at the current level and the level just below the current are associated with supplementary arcs according to the following analysis.

Assume that before the execution of $VirtualGen(V_{i-1}', V_i', V_{i+1}, M_i)$ we have some virtual nodes in V_i', which are associated with supplementary arcs. Then, during the execution of $VirtualGen(V_{i-1}', V_i', V_{i+1}, M_i)$, it is possible that some more virtual nodes will be added to both V_i' and V_{i+1}. Especially, the virtual nodes added to V_{i+1} may also be associated with supplementary arcs. Thus, the virtual nodes in two consecutive levels V_i' and V_{i+1}' can be associated with supplementary arcs. However, by a next call of *VirtualGen*( ), i.e., when executing $VirtualGen(V_i', V_{i+1}', V_{i+2}, M_{i+1})$, any virtual node in V_i' will become covered, or be promoted to V_{i+1}' or V_{i+2} in the sense that a virtual node for it will be created and added to V_{i+1}' or V_{i+2}. Again, only the virtual nodes added to V_{i+2} can be associated with supplementary arcs.

So, the number of virtual arcs maintained in the process is bounded by $O(\kappa \cdot n)$ since the number of supplementary arcs incident to a virtual node is bounded by $O(n)$.

It remains to show how to find a maximal set of node-disjoint paths in $\vec{B}_i \oplus \vec{B}_{i+1}$. For this purpose, we define a maximum flow problem over $\vec{B}_i \oplus \vec{B}_{i+1}$, (with multiple sources and sinks) as follows:

- Each free node in V_{i+1} in B_{i+1} is designated as a *source*. Each free node in V_{i-1}' in B_i is designated as a *sink*.
- Each arc $u \to v$ is associated with a capacity $c(u, v) = 1$. (If nodes u, v are not connected, $c(u, v)$ is considered to be 0.)

  It is a typical 0-1 network. By finding a maximum flow over it we will find a maximum set of node-disjoint paths.

We notice that for each node v in $\vec{B}_i \oplus \vec{B}_{i+1}$, either there is only one arc emanating from it or only one arc entering it. Then, by using Dinic's algorithm [8] for a maximum flow problem over such a 0-1, only $O(\sqrt{n} \cdot m)$ time is required, where n and m are the numbers of the nodes and arcs of the network, respectively. (See pp. 119 – 121 in [15].) Thus, the cost of this task is bounded by

$$O\left(\sqrt{|V_{i-1}'| + |V_i| + |V_i'| + |V_{i+1}|} \cdot \sum_{i=1}^{h}\left(|V_{i-1}'| \cdot |V_i| + |V_i'| \cdot |V_{i+1}|\right)\right.$$

$$\leq O\left(\sqrt{\kappa} \cdot \kappa \cdot n\right).$$

In addition, for each virtual node, once it becomes covered by a maximum matching when it is promoted to a certain level, all the virtual arcs incident to it can be removed. So, the extra space required in bounded by $O(\kappa \cdot n)$.

B. Virtual Node Resolution

After the chain generation, the next step is to resolve (or say, to remove) virtual nodes from chains. For this purpose, we will work top-down along the chains. Two steps will be carried out:

1. Remove virtual nodes, and at the same time connect some nodes according to the connectivity represented by them, and
2. Establish new connections between free nodes by transferring edges along alternating paths within a bipartite graph or cross more than one bipartite graph.

In the first step, we will check virtual nodes level by level, and change G_c (the graph generated during the chain creation) to another graph G' containing part of G's transitive closure, which is necessary to find the final result. In this process, we will first remove all those virtual nodes v with $label(u \to v)$ being the highest (where u is the parent of v along a chain arc), then the virtual nodes with the labels just smaller than v, and so on. Thus, when we try to remove virtual nodes v with $label(u \to v) = i$, all the virtual nodes with higher labels must have been eliminated. In this step, the following operations will be conducted.

i) Let v be a virtual node in V_i'. If v does not have a parent along the corresponding chain, it will be simply removed.

ii) If v has a parent u along a chain with $label(u \to v) = 0$, remove v and connect u to $s(v)$. $label(u \to s(v))$ is set to 0,

iii) If v has a parent u along a chain with $label(u \to v) = i$, remove v and connect u to each reachable node in V_{i-1}.

iv) Construct a combined graph in a way similar to the chain generation, involving the corresponding bipartite graphs, where the direction of each arc corresponding an edge belonging to a maximum matching is reversed.

  See Fig. 12 for illustration.

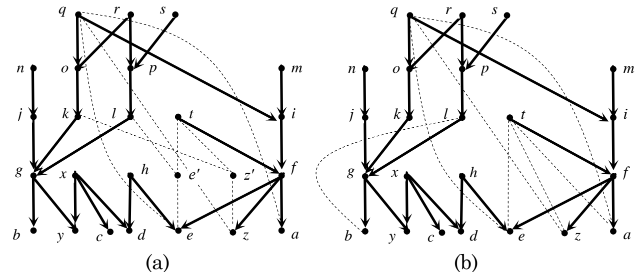


Fig. 12. Illustration for virtual node resolution in G_c

In Fig. 12(a), we show the resulting graph by removing c'' from the graph shown in Fig. 10(e), by which the parent q of c'' along the chain shown in Fig. 11 will be connected to all

those nodes in V_0 that are reachable from q since $label(q \to c'') = 1$. After c'' is eliminated, c' becomes a node without a parent along a chain and is also removed. In Fig. 12(b), e' and z' are continually removed from the graph shown in Fig. 12(a), by which

- l will be connected to b since $label(l \to e') = 1$ and b is the only node in V_0 reachable from l, and
- t will be connected to z since $label(t \to z') = 0$.

After that, we will construct a combined graph as shown in Fig. 13(a), which contains $B(V_1, V_0'; \boldsymbol{C}_1)$ and $B(V_2, V_1'; \boldsymbol{C}_2)$, plus node q, as well as the arcs connecting q, l and t to their respective reachable nodes in V_0 (shown in Fig. 14(b)).

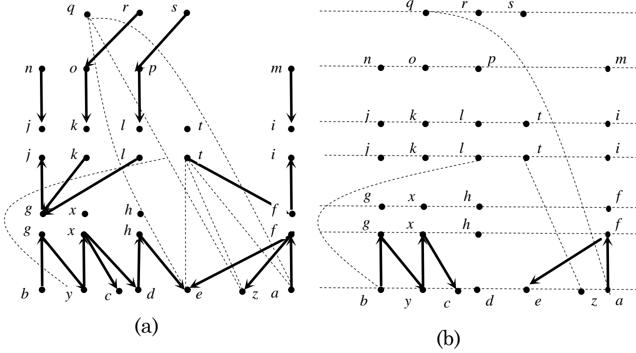


Fig. 13. Illustration for global alternating graphs and node-disjoint paths

We will then find a maximum set of node-disjoint paths with each starting from a node which is a parent of some virtual node along a chain, and ending at a node which is an actual free node relative to M_i in V_{i-1}' in $B(V_i, V_{i-1}'; \boldsymbol{C}_i)$. By transferring the arcs on these paths, we will get the final result. For example, in Fig. 13(b), we can see a possible maximum set of three paths in the graph shown in Fig. 12(a): $P_1 = q \to a \to f \to e$, $P_2 = l \to b \to g \to y \to x \to c$, and $P_3 = t \to z$. Transferring the arcs on each of these paths, we will transform the chains created by the algorithm *GenChain*( ) to a minimum set of chains containing no virtual nodes:

- Along P_1, we will connect node q to node a, cut off $a \to f$ on the corresponding chain, and then connect f to e.
- Along P_2, we connect node l to node b, cut off $b \to g$, connect g to y, cut off $y \to x$, and connect x to c.
- Along P_3, we connect node t to node z.

Fig. 14 demonstrates the final result.

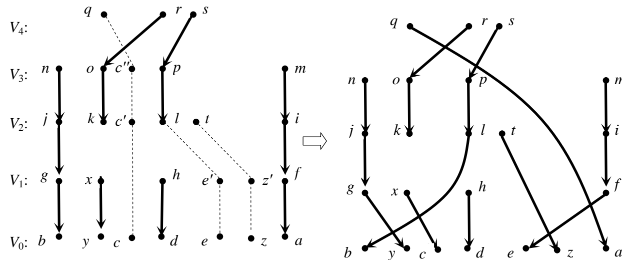


Fig. 14. Transforming the chains generated by Algorithm *GenChain*( ) to the final result

We will repeat the above process to remove all the virtual nodes.

IV. CONCLUSION

In this paper, a new algorithm for finding a minimal decomposition of DAGs is proposed. The algorithm needs O($max\{\sqrt{\kappa}\cdot\kappa\cdot n, \sqrt{n}\cdot m\}$) time and O($\kappa\cdot n$) space, where n and m are the number of the nodes and the arcs in a DAG G, respectively; and κ is the width of G.

REFERENCES

[1] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, Computing a maximum cardinality matching in a bipartite graph in time O(), *Information Processing Letters*, 37(1991), 237 -240.

[2] A.S. Asratian, T. Denley, and R. Haggkvist, *Bipartite Graphs and their Applications*, Cambridge University, 1998.

[3] C. Chekuri and M. Bender, An Efficient Approximation Algorithm for Minimizing Makespan on Uniformly Related Machines, *Journal of Algorithms* 41, 212-224(2001).

[4] Y. Chen and Y.B. Chen, An Efficient Algorithm for Answering Graph Reachability Queries, in *Proc. 24th Int. Conf. on Data Engineering (ICDE 2008)*, IEEE, April 2008, pp. 892-901.

[5] Y. Chen and Y.B. Chen, On the Decomposition of Posets, in *Proc. 2nd Int. Conf. on Computer Science and Service System (CSSS 2012)*, IEEE, Aug. 11-13, Nanjing, China, pp. 1115 - 1119.

[6] Y. Chen and Y.B. Chen, On the Decomposition of Posets into Minimized Set of Node-Disjoint Chains, *2013 Int. Conf. on Computer, Networks and Communication Engineering (ICCNCE 2013),* Beijing, China, May 23-24, 2013, pp. 131-135.

[7] R.P. Dilworth, A decomposition theorem for partially ordered sets, *Ann. Math.* **51** (1950), pp. 161-166.

[8] E.A. Dinic, Algorithm for solution of a problem of maximum flow in a network with power estimation, Soviet Mathematics Doklady, 11(5):1277-1280, 1970.

[9] S. Felsner, L. Wernisch, Maximum *k*-chains in planar point sets: combinatorial structure and algorithms, *SIAM J. Comp.* **28**, 1998, pp. 192-209.

[10] D.R. Fulkerson, Note on Dilworth's embedding theorem for partially ordered sets, *Proc. Amer. Math. Soc.* 7(1956), 701-702.

[11] J.E. Hopcroft, and R.M. Karp, An $n^{2.5}$ algorithm for maximum matching in bipartite graphs, *SIAM J. Comput.* 2(1973), 225-231.

[12] H.V. Jagadish, "A Compression Technique to Materialize Transitive Closure," *ACM Trans. Database Systems*, Vol. 15, No. 4, 1990, pp. 558 - 598.

[13] R.-D. Lou, M. Sarrafzadeh, An optimal algorithm for the maximum two-chain problem, *SIAM J. Disc. Math.* **5**(2), 1992, pp. 285-304.

[14] M.A. Perles, A proof of Dilworth's decomposition theorem for partially ordered sets, *Israel J. of Math.* 1(1963), 105-107.

[15] S. Even, *Graph Algorithms*, Computer Science Press, Inc., Rockville, Maryland, 1979.

[16] H. Goeman, Time and Space Efficient Algorithms for Decomposing Certain Patially Ordered Sets, PhD thesis, Department of Mathematics-Science, Rheinischen Friedrich-Wilhelms Universität Bonn, Germany, Dec. 1999.

[17] R. Tarjan: Depth-first Search and Linear Graph Algorithms, *SIAM J. Compt*. Vol. 1. No. 2. June 1972, pp. 146 -140.

[18] H.S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Commun. ACM* 18, 4 (April 1975), 218 - 220.